Name: Anmol Rastogi Person# 50321153

CSE 565 Fall 2019
HomeWork 2 Report

Setting up Virtual Machine:

1. Upgrade to python 3.7 (https://jcutrer.com/linux/upgrade-python37-ubuntu1810)
    $ sudo apt-get install python3.7
    $ sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.6 1
    $ sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.7 2
    $ sudo update-alternatives --config python3
    !!!! choose 2 (or python3.7)
    $ python3 -V

2. Install pip (https://packaging.python.org/tutorials/installing-packages/#install-pip-setuptools-and-wheel):
    $ wget https://bootstrap.pypa.io/get-pip.py
    $ sudo python3 get-pip.py

3. Install pycryptodome (https://pycryptodome.readthedocs.io/en/latest/src/installation.html):
    $ sudo apt-get install build-essential python3-dev
    $ sudo python3 -m pip install pycryptodomex
    $ python3 -m Cryptodome.SelfTest

Running Code:

$ python3 ComputerSecurityHW2.py testfile.txt

Wrong execution:

$ python3 ComputerSecurityHW2.py

'You must pass a file as argument!!!'

SourceCode:

'''
References:
1. https://pycryptodome.readthedocs.io/en/latest/index.html
2. https://docs.python.org/3/library/hashlib.html
3.https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html
Library Documentation and Library Implementation Code used for reference

Steps to run on VM:

1. Upgrade to python 3.7 (https://jcutrer.com/linux/upgrade-python37-ubuntu1810)
$ sudo apt-get install python3.7
$ sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.6 1
$ sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.7 2
$ sudo update-alternatives --config python3
!!!! choose 2 (or python3.7)
$ python3 -V

2. Install pip (https://packaging.python.org/tutorials/installing-packages/#install-pip-setuptools-and-wheel):
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python3 get-pip.py

3. Install pycryptodome (https://pycryptodome.readthedocs.io/en/latest/src/installation.html):
$ sudo apt-get install build-essential python3-dev
$ sudo python3 -m pip install pycryptodomex
$ python3 -m Cryptodome.SelfTest

'''

```
import json
import os
import struct
import subprocess
import filecmp
import time
import sys
import hashlib
from Cryptodome.Cipher import AES
from Cryptodome.PublicKey import RSA
from Cryptodome.Cipher import PKCS1_OAEP
from Cryptodome.Util.Padding import pad
from Cryptodome.Random import get_random_bytes
from Cryptodome.PublicKey import DSA
from Cryptodome.Signature import DSS
from Cryptodome.Hash import SHA256


class aesCBC:
    def __init__(self):
        self.start_time_key = time.process_time_ns()
        self.key = get_random_bytes(16) # 16bytes*8(bits/bytes)=128bits
        self.end_time_key = time.process_time_ns()
        self.keyperf()
        self.iv = get_random_bytes(AES.block_size)
        self.inputFile = sys.argv[1]
        self.encryptedFile = "encryptedFileAES_CBC" + self.inputFile
        self.decryptedFile = "decryptedFileAES_CBC" + self.inputFile
        self.sizeOfread = 2048


    def keyperf(self):
        print("\n--- %s nanoseconds to generate key using AESNI for CBC mode." %
(self.end_time_key - self.start_time_key))

    def encrypt(self):
        fileSizeBytes = os.path.getsize(self.inputFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        cipher = AES.new(self.key, AES.MODE_CBC, self.iv, use_aesni=True)
        with open(self.encryptedFile, 'wb+') as fout:
```

```
            fout.write(struct.pack('<Q', fileSizeBytes))
            #save IV to output file since it's required during decryption.
            fout.write(self.iv)
        with open(self.inputFile, 'rb') as fileInput, open(self.encryptedFile, 'ab') as fout:
            while True:
                data = fileInput.read(self.sizeOfread)
                length = len(data)
                if length == 0:
                    break
                elif length % 16 != 0:
                    encryptTime = time.process_time_ns()
                    data = pad(data, AES.block_size)
                    diff += (time.process_time_ns() - encryptTime)
                ct_bytes = cipher.encrypt(data)
                fout.write(ct_bytes)
        print("--- %s nanoseconds to encrypt using AESNI in CBC mode." % (diff))
        encryptionSpeed = diff/fileSizeBytes
        print("--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = %s." %
(encryptionSpeed))

    def decrypt(self):
        fileSizeBytes = os.path.getsize(self.encryptedFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        #open encrypted file and read size of encrypted file and IV
        with open(self.encryptedFile, 'rb') as fileInput, open(self.decryptedFile, 'wb+') as fileout:
            fileSize = struct.unpack('<Q', fileInput.read(struct.calcsize('<Q')))[0]
            iv = fileInput.read(AES.block_size)
            cipher = AES.new(self.key, AES.MODE_CBC, self.iv, use_aesni=True )
            #write decrypted file somewhere for verification:
            while True:
                data = fileInput.read(self.sizeOfread)
                #length = len(data)
                if len(data) == 0:
                    break
                decryptTime = time.process_time_ns()
                pt = cipher.decrypt(data)
                diff += (time.process_time_ns() - decryptTime)

                if fileSize > len(pt):
                    fileout.write(pt)
                else:
                    fileout.write(pt[:fileSize])
                # remove padding on last block
                fileSize = fileSize - len(pt)
        print("--- %s nanoseconds to decrypt using AESNI in CBC mode." % (diff))
        decryptionSpeed = diff/fileSizeBytes
        print("--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = %s." %
(decryptionSpeed))
        if filecmp.cmp(self.inputFile, self.decryptedFile):
            print("\nCorrect Encryption and Decryption as Input File \""+ self.inputFile +"\" and
Decrypted file \"" + self.decryptedFile +"\" match.\n")
        else:
```

```
        print("\nIncorrect Encryption and Decryption as Input File \""+ self.inputFile +"\" and
Decrypted file \"" + self.decryptedFile +"\" dont match.\n")


class aesCTR:
    def __init__(self,keysize):
        self.start_time_key = time.process_time_ns()
        self.key = get_random_bytes(keysize)
        self.end_time_key = time.process_time_ns()
        self.keyperf()
        self.nonce = get_random_bytes(8)
        self.inputFile = sys.argv[1]
        self.encryptedFile = "encryptedFileAES_CTR_" + str(keysize*8) + self.inputFile
        self.decryptedFile = "decryptedFileAES_CTR_" + str(keysize*8) + self.inputFile

    def keyperf(self):
        print("--- %s nanoseconds to generate key using AESNI for CTR mode" %
(self.end_time_key - self.start_time_key))

    def encrypt(self):
        fileSizeBytes = os.path.getsize(self.inputFile) #("/Users/anmolrastogi/Documents/Security/
HW/testfile.txt")
        start_time = time.process_time_ns()
        diff = start_time - start_time
        cipher = AES.new(self.key, AES.MODE_CTR, nonce=self.nonce, use_aesni=True)
        #size of file written to output file.
        with open(self.encryptedFile, 'wb+') as fout:
            fout.write(struct.pack('<Q', fileSizeBytes))
        #Since file is encrypted in blocks of multiples of 16 bytes, last block of file might require
padding. so we read 1kb at a time
        with open(self.inputFile, 'rb') as fileInput, open(self.encryptedFile, 'ab') as fout:
            data = fileInput.read(fileSizeBytes)
            encryptTime = time.process_time_ns()
            ct_bytes = cipher.encrypt(data)
            diff += (time.process_time_ns() - encryptTime)
            fout.write(ct_bytes)

        print("--- %s nanoseconds to encrypt using AESNI in CTR mode." % (diff))
        encryptionSpeed = diff/fileSizeBytes
        print("--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = %s." %
(encryptionSpeed))

    def decrypt(self):
        fileSizeBytes = os.path.getsize(self.encryptedFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        #open encrypted file and read size of encrypted file and IV
        with open(self.encryptedFile, 'rb') as fileInput, open(self.decryptedFile, 'wb+') as fileout:
            fileSize = struct.unpack('<Q', fileInput.read(struct.calcsize('<Q')))[0]

            cipher = AES.new(self.key, AES.MODE_CTR, nonce=self.nonce, use_aesni=True )
            #write decrypted file somewhere for verification:
            data = fileInput.read()
            decryptTime = time.process_time_ns()
```

```
            pt = cipher.decrypt(data)
            diff += (time.process_time_ns() - decryptTime)
            fileout.write(pt)
        print("--- %s nanoseconds to decrypt using AESNI in CTR mode." % (diff))
        decryptionSpeed = diff/fileSizeBytes
        print("--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = %s." %
(decryptionSpeed))
        if filecmp.cmp(self.inputFile, self.decryptedFile):
            print("\nCorrect Encryption and Decryption as Input File \""+ self.inputFile +"\" and
Decrypted file \"" + self.decryptedFile +"\" match.\n")
        else:
            print("\nIncorrect Encryption and Decryption as Input File \""+ self.inputFile +"\" and
Decrypted file \"" + self.decryptedFile +"\" dont match.\n")

class hashOfFile:
    def __init__(self): #ch
        self.readSize = 4096
        self.inputFile = sys.argv[1]
        self.hashDigest256 = "hashDigestSHA256"
        self.hashDigest512 = "hashDigestSHA512"
        self.hashDigest3_256 = "hashDigestSHA3_256"

    def sha256(self):
        fileSizeBytes = os.path.getsize(self.inputFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        message = hashlib.sha256()
        with open(self.inputFile, 'rb') as fileInput, open(self.hashDigest256, 'w+') as fileOut:
            while True:
                messageBlock = fileInput.read(self.readSize)
                if len(messageBlock) == 0:
                    break
                hashComputeTime = time.process_time_ns()
                message.update(messageBlock)
                diff += (time.process_time_ns() - hashComputeTime)
            fileOut.write(message.hexdigest())
        print("--- %s nanoseconds to calculate SHA256 of file." % (diff))
        perByteTiming = diff/fileSizeBytes
        print("--- SHA256 per byte speed ((Time Taken To Hash)/(Size of File)) = %s." %
(perByteTiming))

    def sha512(self):
        fileSizeBytes = os.path.getsize(self.inputFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        message = hashlib.sha512()
        with open(self.inputFile, 'rb') as fileInput, open(self.hashDigest512, 'w+') as fileOut:
            while True:
                messageBlock = fileInput.read(self.readSize)
                if len(messageBlock) == 0:
                    break
                hashComputeTime = time.process_time_ns()
                message.update(messageBlock)
                diff += (time.process_time_ns() - hashComputeTime)
```

```
            fileOut.write(message.hexdigest())
        print("--- %s nanoseconds to calculate SHA512 of file." % (diff))
        perByteTiming = diff/fileSizeBytes
        print("--- SHA512 per byte speed ((Time Taken To Hash)/(Size of File)) = %s." %
(perByteTiming))

    def sha3_256(self):
        fileSizeBytes = os.path.getsize(self.inputFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        message = hashlib.sha3_256()
        with open(self.inputFile, 'rb') as fileInput, open(self.hashDigest3_256, 'w+') as fileOut:
            while True:
                messageBlock = fileInput.read(self.readSize)
                if len(messageBlock) == 0:
                    break
                hashComputeTime = time.process_time_ns()
                message.update(messageBlock)
                diff += (time.process_time_ns() - hashComputeTime)
            fileOut.write(message.hexdigest())
        print("--- %s nanoseconds to calculate sha3_256 of file." % (diff))
        perByteTiming = diff/fileSizeBytes
        print("--- sha3_256 per byte speed ((Time Taken To Hash)/(Size of File)) = %s." %
(perByteTiming))

class rsa:
    def __init__(self,keysize):
        self.keysize = keysize
        #self.key = RSA.generate(keysize)
        self.readSize = 127
        self.inputFile = sys.argv[1]
        self.publicKey = "keys/publicKey_"+str(keysize)+".pem"
        self.privateKey = "keys/privateKey_"+str(keysize)+".pem"
        self.rsaEncryptFile = "RSA_"+str(keysize)+"_encrypt_OAEP_Padding" + self.inputFile
        self.rsaDecryptFile = "RSA_"+str(keysize)+"_decrypt_OAEP_Padding" + self.inputFile
        self.storeKeys()
        self.blocksize = "blocksize"

    def storeKeys(self):
        start_time_key = time.time()
        key = RSA.generate(self.keysize)
        end_time_key = time.time()
        private_key = key.export_key()
        #os.mkdir("keys")
        try:
            os.stat("keys")
        except:
            os.mkdir("keys")
        public_key = key.publickey().export_key(pkcs=1)
        with open(self.privateKey,'wb+') as fileOut:
            fileOut.write(private_key)
        with open(self.publicKey,'wb+') as fileOut:
            fileOut.write(public_key)
        subprocess.call(['chmod', '-R', '700', 'keys/'])
```

```
        print("--- %s nanoseconds to generate keys for RSA" % (end_time_key - start_time_key))


    def encrypt(self):
        fileSizeBytes = os.path.getsize(self.inputFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        key = RSA.importKey(open(self.publicKey, 'rb').read())
        cipher = PKCS1_OAEP.new(key)
        with open(self.inputFile, 'rb') as fileInput, open(self.rsaEncryptFile, 'wb+') as fileOut,
open(self.blocksize, 'w+') as blockwrite:
            while True:
                data = fileInput.read(self.readSize)
                if len(data) == 0:
                    break
                encryptTime = time.process_time_ns()
                ct_bytes = cipher.encrypt(data)
                diff += (time.process_time_ns() - encryptTime)
                blockwrite.write(chr(len(ct_bytes)))
                fileOut.write(ct_bytes)
        print("--- %s nanoseconds to encrypt using RSA with OAEP and keysize %d" % (diff,
self.keysize))
        encryptionSpeed = diff/fileSizeBytes
        print("--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = %s." %
(encryptionSpeed))

    def decrypt(self):
        fileSizeBytes = os.path.getsize(self.rsaEncryptFile)
        start_time = time.process_time_ns()
        diff = start_time - start_time
        key = RSA.importKey(open(self.privateKey, 'rb').read())
        cipher = PKCS1_OAEP.new(key)
        with open(self.rsaEncryptFile, 'rb') as fileInput, open(self.rsaDecryptFile, 'wb+') as
fileOut,open(self.blocksize, 'r') as blockread:
            while True:
                length = blockread.read(1)
                if not length:
                    break
                cipherT = fileInput.read(ord(length))#int(self.keysize/8))
                decryptTime = time.process_time_ns()
                plaintext = cipher.decrypt(cipherT)
                diff += (time.process_time_ns() - decryptTime)
                fileOut.write(plaintext[:self.readSize])
        print("--- %s nanoseconds to decrypt using RSA with OAEP and keysize %d" % (diff,
self.keysize))
        decryptionSpeed = diff/fileSizeBytes
        print("--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = %s." %
(decryptionSpeed))
        if filecmp.cmp(self.inputFile, self.rsaDecryptFile):
            print("\nCorrect Encryption and Decryption as Input File \""+ self.inputFile +"\" and
Decrypted file \"" + self.rsaDecryptFile +"\" match.\n")
        else:
            print("\nIncorrect Encryption and Decryption as Input File \""+ self.inputFile +"\" and
Decrypted file \"" + self.rsaDecryptFile +"\" dont match.\n")
```

```
class dsa:

    def __init__(self,keysize):
        self.readSize = 4096
        self.keysize = keysize
        self.start_time_key = time.process_time_ns()
        self.key = DSA.generate(keysize)
        self.end_time_key = time.process_time_ns()
        self.keyperf()
        self.inputFile = sys.argv[1]
        self.publicKey = "dsa_key_"+str(keysize)+".pem"
        self.signature = b''
        self.message = b''
        self.storeKeys()

    def keyperf(self):
        print("\n--- %s nanoseconds to generate DSA key." % (self.end_time_key -
self.start_time_key))

    def storeKeys(self):
        with open(self.publicKey,'wb+') as fileOut:
            fileOut.write(self.key.publickey().export_key())

    def calculateSha(self):
        message = SHA256.new()
        with open(self.inputFile, 'rb') as fileInput:
            while True:
                messageBlock = fileInput.read(self.readSize)
                message.update(messageBlock)
                if len(messageBlock) == 0:
                    break
        return message

    def signMessage(self):
        hash_obj = self.calculateSha()
        signer = DSS.new(self.key, 'fips-186-3')
        signTime = time.process_time_ns()
        self.signature = signer.sign(hash_obj)
        print("--- %s nanoseconds to sign file." % (time.process_time_ns() - signTime))

    def makechange(self):
        with open(self.inputFile, 'ab') as fileOut:
            fileOut.write(b'this line needs to be deleted, why doesnt the hash change here, maybe I
need to stream the creation of hash')

    def verifyMessage(self):
        pub_key = DSA.import_key(open(self.publicKey).read())
        verifier = DSS.new(pub_key, 'fips-186-3')
        hash_obj = self.calculateSha()
        signTime = time.process_time_ns()
        try:
            verifier.verify(hash_obj, self.signature)
            print("The message is authentic")
```

```
        except ValueError:
            print ("The message is not authentic")
        print("--- %s nanoseconds to verify file." % (time.process_time_ns() - signTime))

if len(sys.argv) < 2:
    print ("You must pass a file as argument!!!")
    sys.exit()
#filename = sys.argv[1]



# Using AESNI in CBC Mode:
print("Encryption and Decryption of a file using AES in CBC mode with 128bits key")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_cbc = aesCBC()
print("AES CBC Encryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_cbc.encrypt()
print("AES CBC Decryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_cbc.decrypt()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

# Using AESNI128 in CTR MODE
print("Encryption and Decryption of a file using AES in CTR mode with 128bits key")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_ctr128 = aesCTR(16)
print("AES CTR 128bits Encryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_ctr128.encrypt()
print("AES CTR 128bits Decryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_ctr128.decrypt()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

# Using AESNI256 in CTR MODE
print("Encryption and Decryption of a file using AES in CTR mode with 256bits key")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_ctr128 = aesCTR(32)
print("AES CTR 256bits Encryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_ctr128.encrypt()
print("AES CTR 256bits Decryption")
```

```
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
aes_ctr128.decrypt()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")


#Hash of files:
#sha256
print("Hash of File using SHA256")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
hashSHA256 = hashOfFile()
hashSHA256.sha256()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

print("Hash of File using SHA512")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
hashSHA512 = hashOfFile()
hashSHA512.sha512()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

print("Hash of File using SHA3_256")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
hashSHA3_256 = hashOfFile()
hashSHA3_256.sha3_256()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

print("Encryption and Decryption of a file using 2048bit RSA with PKCS1_OAEP ")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
rsa_2048 = rsa(2048)
print("RSA 2048bit Encryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
rsa_2048.encrypt()
print("RSA 2048bit Decryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
rsa_2048.decrypt()

print("Encryption and Decryption of a file using 3072bit RSA with PKCS1_OAEP ")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
rsa_3072 = rsa(3072)
print("RSA 3072bit Encryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
rsa_3072.encrypt()
```

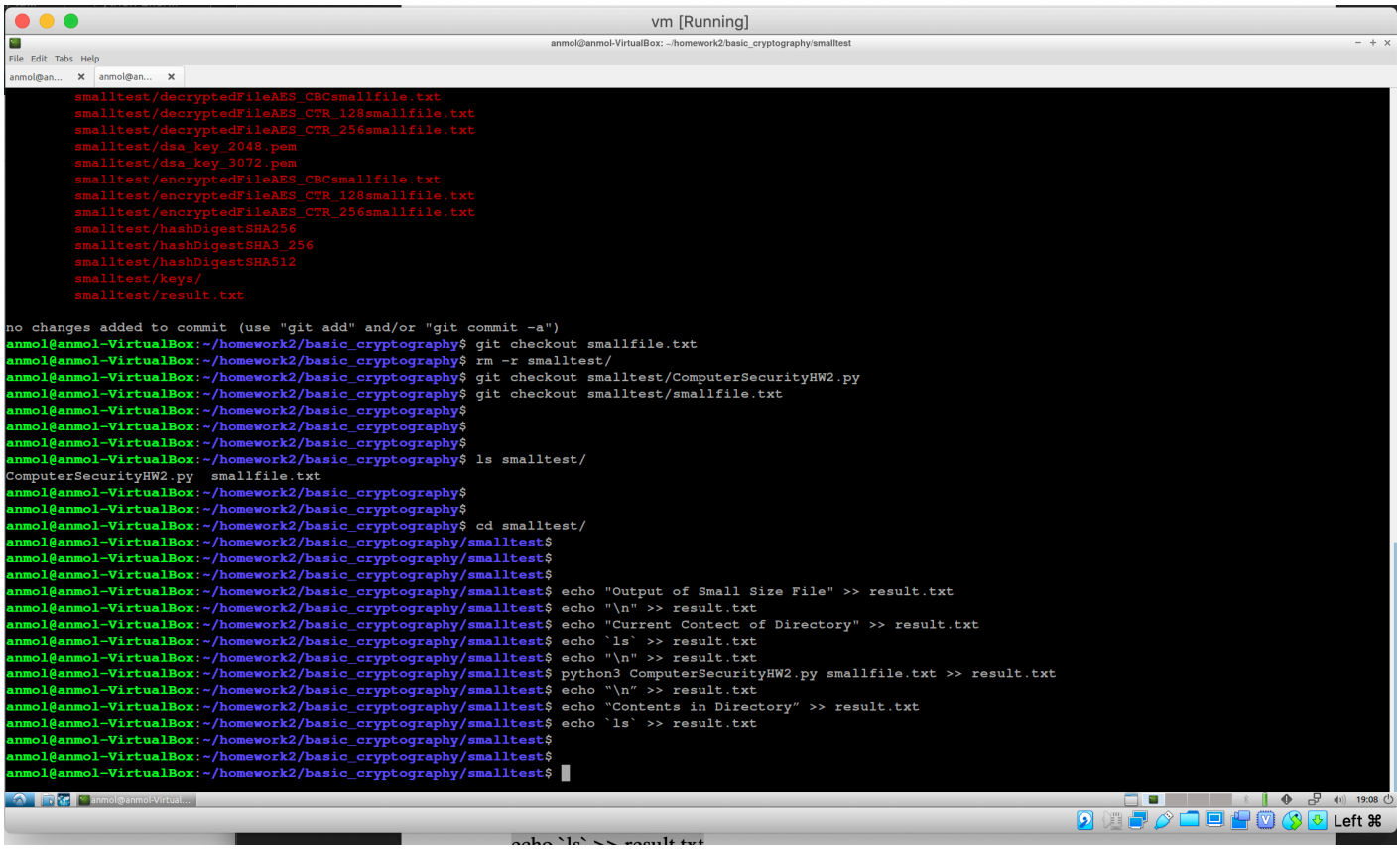
```
print("RSA 2048bit Decryption")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
rsa_3072.decrypt()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

print("Sign and verify a file using 2048bit DSA")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
dsa_2048 = dsa(2048)
print("Sign a file using 2048bit DSA")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
dsa_2048.signMessage()
#dsa_2048.makechange()
print("Verify a file using 2048bit DSA")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
dsa_2048.verifyMessage()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")

print("Sign and verify a file using 3072bit DSA")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
dsa_3072 = dsa(3072)
print("Sign a file using 3072bit DSA")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
dsa_3072.signMessage()
#dsa_3072.makechange()
print("Verify a file using 3072bit DSA")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
dsa_3072.verifyMessage()
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~\n")
```

Name: Anmol Rastogi Person# 50321153

Results of Tests: Small File



>>> cat smalltest/result.txt

Output of Small Size File
\n
Current Contect of Directory
ComputerSecurityHW2.py result.txt smallfile.txt
\n
Encryption and Decryption of a file using AES in CBC mode with 128bits key
~~~~~~

--- 8036 nanoseconds to generate key using AESNI for CBC mode.
AES CBC Encryption
~~~~~~

--- 13077 nanoseconds to encrypt using AESNI in CBC mode.
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 11.572566371681416.
AES CBC Decryption
~~~~~~

--- 28258 nanoseconds to decrypt using AESNI in CBC mode.
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 24.360344827586207.
~~~~~~

Correct Encryption and Decryption as Input File "smallfile.txt" and Decrypted file "decryptedFileAES_CBCsmallfile.txt" match.

~~~~~~

Encryption and Decryption of a file using AES in CTR mode with 128bits key
~~~~~~

--- 3681 nanoseconds to generate key using AESNI for CTR mode
AES CTR 128bits Encryption
~~~~~~

--- 43886 nanoseconds to encrypt using AESNI in CTR mode.
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 38.83716814159292.
AES CTR 128bits Decryption
~~~~~~

--- 16962 nanoseconds to decrypt using AESNI in CTR mode.
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 14.905096660808436.

Correct Encryption and Decryption as Input File "smallfile.txt" and Decrypted file "decryptedFileAES_CTR_128smallfile.txt" match.

~~~~~~

Encryption and Decryption of a file using AES in CTR mode with 256bits key
~~~~~~

--- 3125 nanoseconds to generate key using AESNI for CTR mode
AES CTR 256bits Encryption
~~~~~~

--- 12284 nanoseconds to encrypt using AESNI in CTR mode.
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 10.870796460176992.
AES CTR 256bits Decryption
~~~~~~

--- 10340 nanoseconds to decrypt using AESNI in CTR mode.
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 9.086115992970123.

Correct Encryption and Decryption as Input File "smallfile.txt" and Decrypted file "decryptedFileAES_CTR_256smallfile.txt" match.

~~~~~~

Hash of File using SHA256
~~~~~~

--- 11465 nanoseconds to calculate SHA256 of file.
--- SHA256 per byte speed ((Time Taken To Hash)/(Size of File)) = 10.146017699115044.
~~~~~~

Hash of File using SHA512
~~~~~~

--- 5274 nanoseconds to calculate SHA512 of file.
--- SHA512 per byte speed ((Time Taken To Hash)/(Size of File)) = 4.667256637168141.
~~~~~~

Hash of File using SHA3_256
~~~~~~

--- 10792 nanoseconds to calculate sha3_256 of file.
--- sha3_256 per byte speed ((Time Taken To Hash)/(Size of File)) = 9.550442477876107.
~~~~~~

Encryption and Decryption of a file using 2048bit RSA with PKCS1_OAEP
~~~~~~

--- 0.19330215454101562 nanoseconds to generate keys for RSA
RSA 2048bit Encryption
~~~~~~

--- 3673474 nanoseconds to encrypt using RSA with OAEP and keysize 2048
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 3250.861946902655.
RSA 2048bit Decryption
~~~~~~

--- 10773621 nanoseconds to decrypt using RSA with OAEP and keysize 2048
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 4676.05078125.

Correct Encryption and Decryption as Input File "smallfile.txt" and Decrypted file
"RSA_2048_decrypt_OAEP_Paddingsmallfile.txt" match.

Encryption and Decryption of a file using 3072bit RSA with PKCS1_OAEP
~~~~~~

--- 0.4758162498474121 nanoseconds to generate keys for RSA
RSA 3072bit Encryption
~~~~~~

--- 5811126 nanoseconds to encrypt using RSA with OAEP and keysize 3072
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 5142.589380530973.
RSA 2048bit Decryption
~~~~~~

--- 25312716 nanoseconds to decrypt using RSA with OAEP and keysize 3072
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 7324.28125.

Correct Encryption and Decryption as Input File "smallfile.txt" and Decrypted file
"RSA_3072_decrypt_OAEP_Paddingsmallfile.txt" match.

~~~~~~

Sign and verify a file using 2048bit DSA
~~~~~~
~~~~~~

--- 2763603045 nanoseconds to generate DSA key.
Sign a file using 2048bit DSA
~~~~~~

--- 434826 nanoseconds to sign file.
Verify a file using 2048bit DSA
~~~~~~

The message is authentic
--- 706680 nanoseconds to verify file.
~~~~~~

Sign and verify a file using 3072bit DSA
~~~~~~

--- 1811754055 nanoseconds to generate DSA key.
Sign a file using 3072bit DSA
~~~~~~

--- 830592 nanoseconds to sign file.
Verify a file using 3072bit DSA
~~~~~~

The message is authentic
--- nanoseconds to verify file.
~~~~~~

“n”
“Contents in Directory”
blocksize ComputerSecurityHW2.py decryptedFileAES_CBCsmallfile.txt
decryptedFileAES_CTR_128smallfile.txt decryptedFileAES_CTR_256smallfile.txt
dsa_key_2048.pem dsa_key_3072.pem encryptedFileAES_CBCsmallfile.txt
encryptedFileAES_CTR_128smallfile.txt encryptedFileAES_CTR_256smallfile.txt
hashDigestSHA256 hashDigestSHA3_256 hashDigestSHA512 keys result.txt
RSA_2048_decrypt_OAEP_Paddingsmallfile.txt RSA_2048_encrypt_OAEP_Paddingsmallfile.txt
RSA_3072_decrypt_OAEP_Paddingsmallfile.txt RSA_3072_encrypt_OAEP_Paddingsmallfile.txt
smallfile.txt

A) Small File

1. Small file ‘smallfile.txt’ of 1.1K
2. The complete output of the program can be found in the ‘smalltest’ directory of zip file under the name ‘result.txt’

Function	Time (In NanoSeconds)	Performance (nanosec/byte)

AESNI CBC MODE (128bits key)

Key Generation	8036	
~~~~~~

Encryption	13077	11.5725
Decryption	20258	24.3603

AESNI CTR MODE (128 bits key)

Key Generation	3681	
Encryption	43886	38.8371
Decryption	16962	14.9050

AESNI CTR MODE (256 bits key)

Key Generation	3125	
Encryption	12284	10.870
Decryption	10340	9.086

RSA 2048 bit

Key Generation	0.1933	
Encryption	3673474	3250.8619
Decryption	10773621	4676.0507

RSA 3072 bit

Key Generation	0.47581	
Encryption	5811126	5142.5893
Decryption	25312716	7324.28125

Hash of File

Hash Function	Time (in nanosecs)	Performance(time/size of file)
SHA 256	11465	10.1460
SHA 512	5274	4.6672
SHA3_256	10792	9.55042

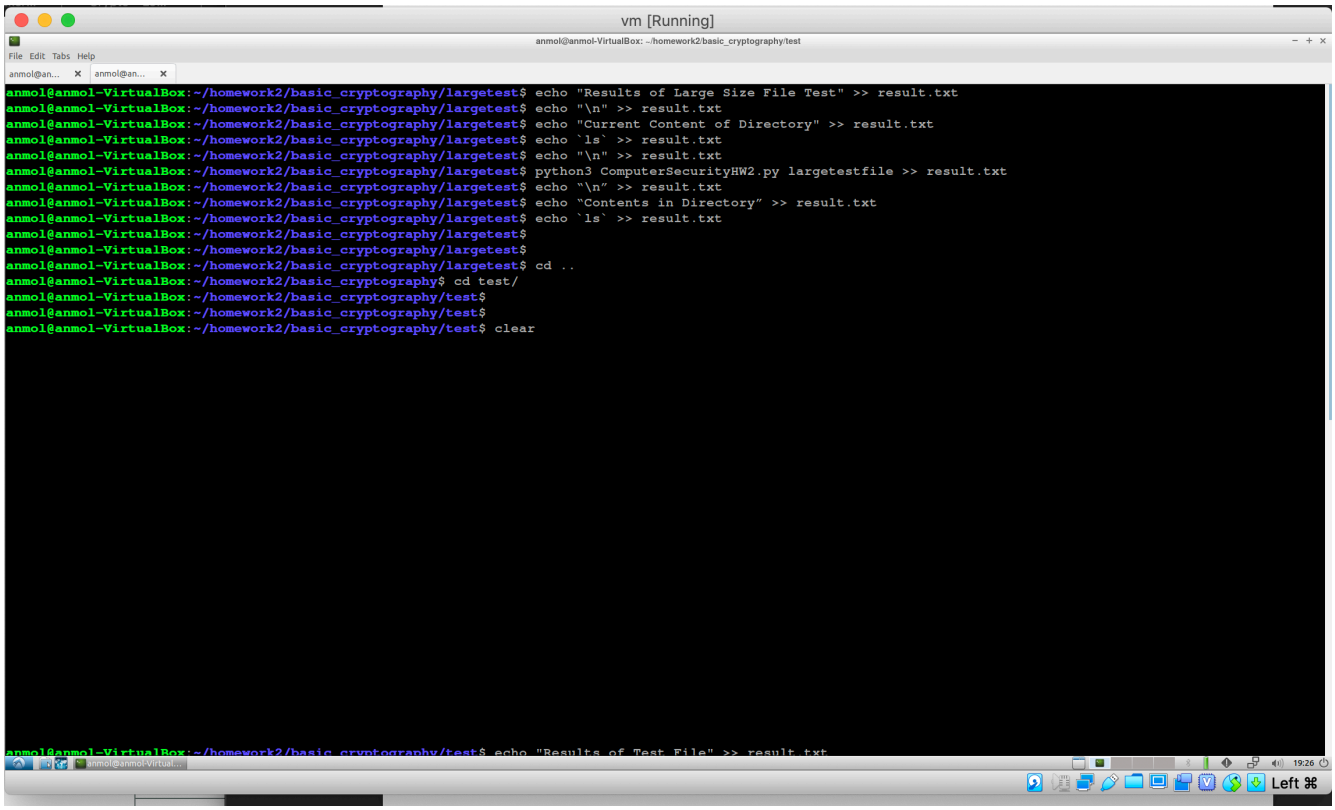
DSA 2048 bit

Function	Time
Key Generation	2763603045
Sign	434826
Verify	706680

DSA 3072 bit

Function	Time
Key Generation	1811754055
Sign	830592
Verify	1398793

Results of Tests: Large File



>>> cat ./largetest/result.txt

Results of Test File
\n
Current Content of Directory
ComputerSecurityHW2.py largetestfile
\n
Encryption and Decryption of a file using AES in CBC mode with 128bits key
~~~~~~

--- 4988 nanoseconds to generate key using AESNI for CBC mode.
AES CBC Encryption
~~~~~~

--- 19700 nanoseconds to encrypt using AESNI in CBC mode.
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) =
0.0018855541380399827.
AES CBC Decryption
~~~~~~

--- 120218275 nanoseconds to decrypt using AESNI in CBC mode.
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 11.506457855246644.
~~~~~~

Correct Encryption and Decryption as Input File "largetestfile" and Decrypted file "decryptedFileAES_CBClargetestfile" match.

~~~~~~

Encryption and Decryption of a file using AES in CTR mode with 128bits key
~~~~~~

--- 7466 nanoseconds to generate key using AESNI for CTR mode
AES CTR 128bits Encryption
~~~~~~

--- 29366560 nanoseconds to encrypt using AESNI in CTR mode.
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 2.810773539492357.
AES CTR 128bits Decryption
~~~~~~

--- 25687228 nanoseconds to decrypt using AESNI in CTR mode.
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 2.4586102519509967.

Correct Encryption and Decryption as Input File "largetestfile" and Decrypted file "decryptedFileAES_CTR_128largetestfile" match.

~~~~~~

Encryption and Decryption of a file using AES in CTR mode with 256bits key
~~~~~~

--- 7110 nanoseconds to generate key using AESNI for CTR mode
AES CTR 256bits Encryption
~~~~~~

--- 28620109 nanoseconds to encrypt using AESNI in CTR mode.
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 2.739328170360678.
AES CTR 256bits Decryption
~~~~~~

--- 25749920 nanoseconds to decrypt using AESNI in CTR mode.
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 2.464610712332137.

Correct Encryption and Decryption as Input File "largetestfile" and Decrypted file "decryptedFileAES_CTR_256largetestfile" match.

~~~~~~

Hash of File using SHA256
~~~~~~

--- 27367306 nanoseconds to calculate SHA256 of file.
--- SHA256 per byte speed ((Time Taken To Hash)/(Size of File)) = 2.6194181256500735.
~~~~~~

Hash of File using SHA512
~~~~~~

--- 17501601 nanoseconds to calculate SHA512 of file.
--- SHA512 per byte speed ((Time Taken To Hash)/(Size of File)) = 1.6751378775570913.
~~~~~~

Hash of File using SHA3_256
~~~~~~

--- 28272781 nanoseconds to calculate sha3_256 of file.
--- sha3_256 per byte speed ((Time Taken To Hash)/(Size of File)) = 2.7060842237790963.
~~~~~~

Encryption and Decryption of a file using 2048bit RSA with PKCS1_OAEP
~~~~~~

--- 0.2800004482269287 nanoseconds to generate keys for RSA
RSA 2048bit Encryption
~~~~~~

--- 31183345833 nanoseconds to encrypt using RSA with OAEP and keysize 2048
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 2984.6643032154825.
RSA 2048bit Decryption
~~~~~~

--- 98516566900 nanoseconds to decrypt using RSA with OAEP and keysize 2048
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 4677.821477057933.

Correct Encryption and Decryption as Input File "largetestfile" and Decrypted file
"RSA_2048_decrypt_OAEP_Paddinglargetestfile" match.

Encryption and Decryption of a file using 3072bit RSA with PKCS1_OAEP
~~~~~~

--- 1.1687757968902588 nanoseconds to generate keys for RSA
RSA 3072bit Encryption
~~~~~~

--- 52689018980 nanoseconds to encrypt using RSA with OAEP and keysize 3072
--- Encryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 5043.045571929249.
RSA 2048bit Decryption
~~~~~~

--- 238548181136 nanoseconds to decrypt using RSA with OAEP and keysize 3072
--- Decryption Speed per byte ((Time Taken To Encrypt)/(Size of File)) = 7551.256539175287.

Correct Encryption and Decryption as Input File "largetestfile" and Decrypted file
"RSA_3072_decrypt_OAEP_Paddinglargetestfile" match.

~~~~~~

Sign and verify a file using 2048bit DSA
~~~~~~
~~~~~~

--- 1557115886 nanoseconds to generate DSA key.
Sign a file using 2048bit DSA
~~~~~~

--- 440181 nanoseconds to sign file.
Verify a file using 2048bit DSA
~~~~~~

The message is authentic
--- 707241 nanoseconds to verify file.
~~~~~~

Sign and verify a file using 3072bit DSA
~~~~~~

--- 18685160018 nanoseconds to generate DSA key.
Sign a file using 3072bit DSA
~~~~~~

--- 1277562 nanoseconds to sign file.
Verify a file using 3072bit DSA
~~~~~~

The message is authentic
--- 2277531 nanoseconds to verify file.
~~~~~~

“n”
“Contents in Directory”
blocksize ComputerSecurityHW2.py decryptedFileAES_CBClargetestfile
decryptedFileAES_CTR_128largetestfile decryptedFileAES_CTR_256largetestfile
dsa_key_2048.pem dsa_key_3072.pem encryptedFileAES_CBClargetestfile
encryptedFileAES_CTR_128largetestfile encryptedFileAES_CTR_256largetestfile
hashDigestSHA256 hashDigestSHA3_256 hashDigestSHA512 keys largetestfile result.txt
RSA_2048_decrypt_OAEP_Paddinglargetestfile RSA_2048_encrypt_OAEP_Paddinglargetestfile
RSA_3072_decrypt_OAEP_Paddinglargetestfile RSA_3072_encrypt_OAEP_Paddinglargetestfile

---

Large File:

1. Small file ‘largetestfile’ of 10M
2. The complete output of the program can be found in the ‘largetest’ directory of zip file under the name ‘result.txt’

Function	Time (In NanoSeconds)	Performance (nanosec/byte)

AESNI CBC MODE (128bits key)

Key Generation	4988	
~~~~~~

Encryption	19700	0.00188
Decryption	120218275	11.5064

AESNI CTR MODE (128 bits key)

Key Generation	7466	
Encryption	29366560	2.8107
Decryption	25687228	2.4586

AESNI CTR MODE (256 bits key)

Key Generation	7110	
Encryption	28620109	2.7393
Decryption	25749920	2.4646

RSA 2048 bit

Key Generation	0.2800	
Encryption	31183345833	2984.6643
Decryption	98516566900	4677.8214

RSA 3072 bit

Key Generation	1.16877	
Encryption	52689018980	5043.0455
Decryption	238548181136	7551.2565

Hash of File

Hash Function	Time (in nanosecs)	Performance(time/size of file)
SHA 256	27367306	2.6194
SHA 512	17501601	1.6751
SHA3_256	28272781	2.7060

DSA 2048 bit

Function	Time
Key Generation	1557115886
Sign	440181
Verify	707241

DSA 3072 bit

Function	Time
Key Generation	18685160018
Sign	1277562
Verify	2277531

How per byte speed changes for different algorithms between small and large files?

All times are in nanoseconds

For Encryption:

The byte per second data for encryption:

	Small File	**Large File**
AESNI CBC	11.5725	0.00188
AESNI CTR 128	38.8371	2.8107
AESNI CTR 256	10.870	2.7393
RSA 2048	3250.8619	2984.6643
RSA 3072	5142.5893	5043.0455

The byte per second is significantly lower for large file in this program. Hence for this implementation it takes slightly longer to encrypt file of 10k times magnitude.

For Decryption:

The byte per second data for decryption:

	Small File	**Large File**
AESNI CBC	24.3603	11.5064
AESNI CTR 128	14.905	2.4586
AESNI CTR 256	9.086	2.4646

RSA 2048	4676.0507	4677.8214
RSA 3072	7324.28125	7551.2565

During Decryption the table shows for Large files byte per second is significantly lower for AESNI operations, but it’s slightly higher for RSA operations.

How encryption and decryption times differ for a given encryption algorithm?

All times are in nanoseconds

1. AESNI CBC:

Small File:

Encryption	13077
Decryption	20258

Large File:

Encryption	19700
Decryption	120218275

The data shows it take less time to encrypt than it takes to decrypt even with hardware implementation of AES.

2. AESNI CTR

128bits

Small File:

Encryption	43886
Decryption	16962

Large File:

Encryption	29366560
Decryption	25687228

256bits

Small File:

Encryption	12284
Decryption	10340

Large File:

Encryption	28620109
Decryption	25749920

The data shows for AESNI in CTR mode it consistently took less time to decrypt than to encrypt.

3. RSA

2048bit

Small File:

Encryption	3673474
Decryption	10773621

Large File:

Encryption	31183345833
Decryption	98516566900

3072bit

Small File:

Encryption	5811126
Decryption	25312716

Large File:

Encryption	52689018980
Decryption	238548181136

The data shows it take less time to encrypt than it takes for decrypting in RSA.

How key generation times differ with the increase in the key size

All times are in nanoseconds

Key Generation Times:

	Small File	**Large File**
AESNI CBC	8036	4988
AESNI CTR 128	3681	7466
AESNI CTR 256	3125	7110
RSA 2048	0.1933	0.2800
RSA 3072	0.47581	1.16877
DSA 2048	2763603045	1557115886

DSA 3072	1811754055	1557115886

Observations:
1. For AESNI in CTR mode it takes a shorter time generate the longer key.
2. RSA key generation is very fast for the key sizes, and larger key sizes take longer time to generate.
3. In DSA the longer key takes lesser time to generate.

How hashing time differs between the algorithms and with increase of the hash size.

All times are in nanoseconds

Hashing Comparisons:

	Small File	**Large File**
SHA 256	11465	27367306
SHA 512	5274	17501601
SHA3_256	10792	28272781

Observations:
1. For same length of data speed of hashing algorithm are: SHA 512 >> SHA3_256 > SHA3_256. SHA 512 is significantly faster than the other two algorithms.
2. Time to perform Hash with any algorithm is directly proportional to the size of the file.

How performance of symmetric key encryption (AES), hash functions, and public-key encryption (RSA) compare to each other

All times are in nanoseconds

1. Key Generation:

	Small File	**Large File**
AESNI CBC	8036	4988
AESNI CTR 128	3681	7466
AESNI CTR 256	3125	7110
RSA 2048	0.1933	0.2800
RSA 3072	0.47581	1.16877

2. Encryption:

	Small File	Large File
AESNI CBC	13077	19700
AESNI CTR 128	43886	29366560
AESNI CTR 256	12284	28620109
RSA 2048	3673474	31183345833
RSA 3072	5811126	52689018980

3. Decryption:

	Small File	Large File
AESNI CBC	20258	120218275
AESNI CTR 128	16962	25687228
AESNI CTR 256	10340	25749920
RSA 2048	10773621	98516566900
RSA 3072	25312716	238548181136

4. Hashing Comparisons:

	Small File	Large File
SHA 256	11465	27367306
SHA 512	5274	17501601
SHA3_256	10792	28272781

Observations:
1. Key Generation is very fast in public-key encryption compared to symmetric key encryption(AES)
2. Encryption/Decryption using AESNI is faster than both finding hash of a file or encrypting/ decrypting using RSA.
3. RSA is slower since it is better suited for encrypting small data, numbers, encryption keys, passwords than larger data such as files. It's mostly employed to safely transmit small secrets safely rather than encrypt data.